(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
17 August 2006 (17.08.2006)

PCT

(10) International Publication Number
WO 2006/085103 A1

(51) International Patent Classification:
G06F 21/22 (2006.01)

(21) International Application Number:
PCT/GB2006/000483

(22) International Filing Date:
13 February 2006 (13.02.2006)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/652,019 11 February 2005 (11.02.2005) US

(71) Applicant *(for all designated States except US)*: **SIMPLEX MAJOR SDN, BHD.** [MY/MY]; Unit No.203, 2nd Floor, Block C, Damansara Intan No.1, Jalan SS20/27, 47400 Petaling Jaya, Selangor Darul Ehsan (MY).

(72) Inventor; and

(75) Inventor/Applicant *(for US only)*: **SAFA, John, Aram** [GB/GB]; 34 Lenton Road, The Park Estate, Nottingham NG7 1DU (GB).

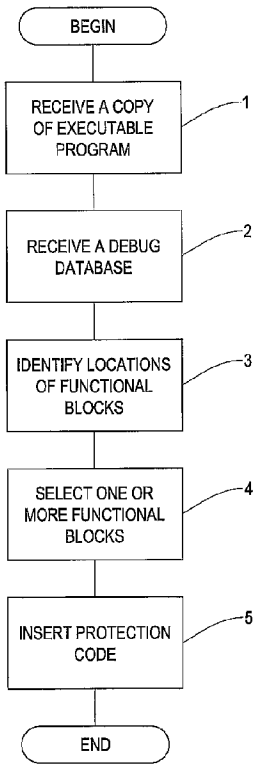
(74) Agent: **FORRESTER KETTLEY & CO**; Forrester House, 52 Bounds Green Road, London N11 2EY (GB).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, LY, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SM, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

[Continued on next page]

(54) Title: SOFTWARE PROTECTION METHOD



(57) Abstract: A method of protecting an executable program from reverse engineering and/or tampering. The method includes receiving a copy of the executable program together with a debug database, the database storing the locations of functional blocks within the executable program. Protection code is inserted into the executable program so as to overwrite at least part of a functional block of the executable program. Subsequent execution of the functional block causes the protection code to be executed. The protection code, when executed, performs an operation and executes a copy of the overwritten part of the functional block.

WO 2006/085103 A1

Published:
— *with international search report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

SOFTWARE PROTECTION METHOD

The present invention relates to a software protection method, and in particular to a method of protecting an executable program which inhibits reverse engineering and/or tampering of the program.

Methods of software protection are continually being sought to inhibit the illegal copying and distribution of software. One form of software protection involves embedding protection code within a software application. Upon executing the software application, the protection code determines whether or not the computer attempting to execute the application is authorised to do so. Unfortunately, this form of software protection is susceptible to attacks by debuggers and disassemblers. For example, the software application may be disassembled and the location of protection code identified from the assembly code. Once the location of the protection code is known, additional code can be inserted into the software application to avoid the protection code or to fool the protection code into believe that the computer is authorised to execute the software application.

Methods of protecting a software application against reverse engineering include encrypting one or more fragments of the application. However, this requires knowledge of the source code of the application since not all fragments of the application are suitable for encryption. For example, any direct jump or call instruction that points to a location within encrypted code will naturally result in a fatal error during execution. Additionally, if no regard is given to the location of the code being encrypted, instructions immediately preceding the encrypted code may be incorrectly executed. Furthermore, the contents of registers are likely to be overwritten during decryption. Consequently, any routine that is partly encrypted is unlikely to execute correctly. Such methods of protection are therefore unsuitable for software applications for which the source code is unknown.

In addition to illegal copying and distribution of software, a further concern is the infection of software by malicious code, such as virus and phishing code, particularly in view of the increasing exchange of data over communication networks. Whilst there are anti-virus applications that scan software for the presence of virus code, new viruses are continually being released. Consequently, anti-virus applications require frequent updating in order to be of any practical use.

In a first aspect, the present invention provides a method of protecting an executable program, the method comprising: receiving a copy of the executable program, the executable program comprising executable instructions arranged as a plurality of functional blocks; receiving a debug database associated with the executable program, the debug database storing the locations of the functional blocks within the executable program; identifying from the debug database the location of a functional block within the executable program; and inserting protection code into the executable program, the protection code overwriting at least part of the functional block, wherein execution of the functional block causes the protection code to be executed, and execution of the protection code performs an operation and executes a copy of the at least part of the functional block overwritten by the protection code.

Preferably, inserting protection code comprises: encoding at least part of the functional block to create encoded instructions; inserting a decoding routine into the executable program; and replacing the at least part of the functional block of the executable program with a call instruction to the decoding routine and with the encoded instructions, wherein the decoding routine comprises executable instructions for decoding the encoded instructions and for executing the decoded instructions.

Advantageously, encoding comprises encrypting the at least part of the functional block using an encryption key, and decoding comprises decrypting the encoded instructions using a decryption key.

Conveniently, the encryption key is derived from a fragment of code of the executable program, the decoding routine derives the decryption key from the fragment of code, and the encoded instructions are successfully decrypted in the event that no corruption of the fragment of code has occurred.

Preferably, the encryption key is derived from the instructions of a further functional block, the further functional block being encoded, and the decoding routine decodes the further functional block and derives the decryption key from the instructions of the further functional block, and the encoded instructions are successfully decrypted in the event that the further functional block is successfully decrypted.

Advantageously, at least one functional block includes a call or jump instruction to a location within a further function block, and the debug database stores the location to which each call or jump instruction of the executable program points, and the step of encoding a functional block comprises: selecting at least part of a functional block for encoding; identifying from the contents of the debug database jump and call instructions that point to a location within the selected at least part of the functional block; and encoding the selected at least part of the functional block in the event that no jump and calls instructions are identified.

Conveniently, inserting protection code comprises: encoding at least part of each of a plurality of functional blocks to create a plurality of segments of encoded instructions, each segment corresponding to a respective functional block; inserting a decoding routine into the executable program; and replacing the at least part of each of the plurality of functional blocks with a call instruction to the decoding routine and with a respective segment of encoded instructions, wherein

the decoding routine comprises executable instructions for decoding and executing encoded instructions.

Preferably, inserting protection code comprises: encoding at least part of a first functional block with a first encoding algorithm to create first encoded instructions; encoding at least part of a second functional block with a second encoding algorithm to create second encoded instructions; inserting a first decoding routine into the executable program; inserting a second decoding routine into the executable program; replacing the at least part of the first functional block with a call instruction to the first decoding routine and with the first encoded instructions; and replacing the at least part of the second functional block of the executable program with a call instruction to the second decoding routine and with the second encoded instructions, wherein the first decoding routine comprises executable instructions for decoding the first encoded instructions and for executing the first decoded instructions, and the second decoding routine comprises executable instructions for decoding the second encoded instructions and for executing the second decoded instructions.

Advantageously, the first encoding routine and second encoding routine are different.

Conveniently, the method further comprises disassembling the functional block and selecting one or more instructions from the disassembled code, and wherein encoding at least part of the functional block comprises encoding one or more fragments of the functional block corresponding to the selected instructions.

Preferably, the decoding routine is inserted into the executable program at locations occupied by redundant instructions.

Advantageously, the debug database stores the start address of each functional block and information regarding the end address of each functional block, and the method comprises comparing the end address of a functional block with the start address of an adjacent functional block to identify redundant instructions between the two functional blocks.

Conveniently, the method comprises disassembling a functional block, and analysing the disassembled code to identify redundant instructions within the functional block.

Preferably, the functional block includes at least one call instruction having a call address, and the method further comprises: inserting a verification routine into the executable program; and modifying the call address of the call instruction such that the verification routine is executed prior to calling the call address.

Advantageously, the debug database stores the locations of call instructions within the executable program and inserting protection code comprises: inserting a verification routine into the executable program; identifying from the debug database the location of a call instruction within the executable program, the call instruction having a call address; and modifying the call address of the call instruction such that the verification routine is executed prior to calling the call address.

Conveniently, the verification routine comprises instructions for detecting corruption of the executable program.

Preferably, the verification routine comprises instructions for calling the call address in the event that no corruption is detected.

Advantageously, at least a fragment of the executable program is stored in a memory device during execution and the verification routine detects corruption of the fragment stored in memory.

Conveniently, the verification routine comprises instructions for corrupting or deleting the fragment of the executable program stored in the memory device in the event that corruption is detected.

Preferably, the verification routine comprises instructions for terminating execution of the executable program in the event that the corruption is detected.

Advantageously, the verification routine employs a CRC algorithm.

Conveniently, the verification routine comprises instructions for detecting whether a debug process is executing and for terminating execution of the executable program in the event that a debug process is detected.

Preferably, the method further comprises inserting intermediary code into the executable program, the intermediary code comprising a first call instruction to the verification routine and a second call instruction to the call address, and the call address is modified to call the intermediary code.

Advantageously, the method comprises identifying from the debug database the location of a plurality of call instructions within the executable program, each of the plurality of call instructions having a call address; inserting a respective segment of intermediary code for each of the plurality of call instructions, each segment of intermediary code comprising a first call instruction to the verification routine and a second call instruction to the call address of a respective call instruction of the executable program; and modifying the call address of each of

the plurality of call instructions of the executable program to call a respective segment of intermediary code.

In a second aspect, the present invention provides a method of protecting an executable program, the method comprising: receiving a copy of the executable program, the executable program comprising executable instructions arranged as a plurality of functional blocks; receiving a debug database associated with the executable program, the debug database storing the locations of the functional blocks within the executable program; identifying from the debug database the location of a functional block within the executable program; encoding at least part of the functional block to create encoded instructions; inserting a decoding routine into the executable program; and replacing the at least part of the functional block of the executable program with a call instruction to the decoding routine and with the encoded instructions, wherein the decoding routine comprises executable instructions for decoding the encoded instructions and for executing the decoded instructions.

In a third aspect, the present invention provides a method of protecting an executable program, the method comprising: receiving a copy of the executable program; receiving a debug database associated with the executable program, the debug database storing the locations of call instructions within the executable program; inserting a verification routine into the executable program; identifying from the debug database the location of a call instruction within the executable program, the call instruction having a call address; and modifying the call address of the call instruction such that the verification routine is executed prior to calling the call address.

In a fourth aspect, the present invention provides a device for protecting an executable program, the device comprising: means for receiving a copy of the executable program, the executable program comprising executable instructions arranged as a plurality of functional blocks; means for receiving a debug

database associated with the executable program, the debug database storing the locations of the functional blocks within the executable program; means for identifying from the debug database the location of a functional block within the executable program; and means for inserting protection code into the executable program, the protection code overwriting at least part of the functional block, wherein execution of the functional block causes the protection code to be executed, and execution of the protection code performs an operation and executes a copy of the at least part of the functional block overwritten by the protection code.

In a fifth aspect, the present invention provides a device for protecting an executable program, the device comprising: means for receiving a copy of the executable program, the executable program comprising executable instructions arranged as a plurality of functional blocks; means for receiving a debug database associated with the executable program, the debug database storing the locations of the functional blocks within the executable program; means for identifying from the debug database the location of a functional block within the executable program; means for encoding at least part of the functional block to create encoded instructions; means for inserting a decoding routine into the executable program; and means for replacing the at least part of the functional block of the executable program with a call instruction to the decoding routine and with the encoded instructions, wherein the decoding routine comprises executable instructions for decoding the encoded instructions and for executing the decoded instructions.

In a sixth aspect, the present invention provides a device for protecting an executable program, the device comprising: means for receiving a copy of the executable program; means for receiving a debug database associated with the executable program, the debug database storing the locations of call instructions within the executable program; means for inserting a verification routine into the executable program; means for identifying from the debug database the location

of a call instruction within the executable program, the call instruction having a call address; and means for modifying the call address of the call instruction such that the verification routine is executed prior to calling the call address.

In a seventh aspect, the present invention provides a computer comprising instructions for performing the aforementioned method.

In an eighth aspect, the present invention provides a computer program or suite of computer programs executable by a computer to perform the aforementioned method.

In a ninth aspect, the present invention provides a computer program product storing computer program code executable by a computer to perform the aforementioned method.

In order that the present invention may be more readily understood, embodiments thereof will now be described, by way of example, with reference to the accompanying drawings, in which:

Figure 1 illustrates a flow diagram of a method of protecting an executable program embodying the present invention;

Figure 2 illustrates a flow diagram of a method of inserting protection code into an executable program in accordance with a first embodiment of the present invention;

Figure 3 illustrates an executable program before and after insertion of protection code by the method of Figure 2;

Figure 4 illustrates a flow diagram of a method of inserting protection code into an executable program in accordance with a second embodiment of the present invention; and

Figure 5 illustrates an executable program before and after insertion of protection code by the method of Figure 4.

Figure 1 illustrates an embodiment of the present invention in which protection code is inserted into an executable program to provide a protected copy of the executable program.

The method comprises receiving 1 a copy of the executable program to be protected, receiving 2 a debug database associated with the executable program, identifying 3 from the contents of the debug database the locations of functional blocks within the executable program, selecting 4 one or more functional blocks of the executable program, and inserting 5 protection code into the executable program such that at least a part of each of the selected functional blocks is overwritten.

A functional block is regarded as a block of one or more executable instructions which, when executed, perform a particular function.

For a simple executable program (i.e. a program having few functional blocks and/or relatively simple instructions), it may be possible to identify functional blocks within the executable program by disassembling the executable program and analysing the assembly code. However, disassembly does not recover any symbolic information such as original function or class names for object-orientated languages such as C++ which compile directly to machine code (conversely, newer languages such as Java & C# retain symbolic information when compiled to their intermediate or byte code forms). This lack of symbolic

information makes it extremely difficult to identify the start and end locations of functional blocks within more complicated executable programs and relate those back to the source code. This is particularly true of programs executed on platforms having an instruction set that employs variable length instructions, e.g. x86 instruction set. This difficulty in identifying functional blocks is resolved by the provision of a debug database associated with the executable program, and identifying 3 from the contents of the debug database the locations of the functional blocks.

The debug database is generated by a linker, typically by selecting an option to collate debug information, when assembling objects to create the executable program. The debug database includes, among other things, the relative address (i.e. location) of each functional block within the executable program. Additionally, the debug database includes the address of each direct call and jump instruction within the executable program, as well as the address to which the call or jump instruction points.

By way of example only, Microsoft® Visual Studio® can be configured to output a debug database file called a program database (PDB) file. The PDB file stores, among other things, the addresses of the functional blocks, the addresses of direct call and jump instructions and the addresses to which the call and jump instructions point.

Two embodiments for inserting protection code into the executable program will now be described.

In a first embodiment, as illustrated in Figure 2, a verification routine is inserted 6 into the executable program. The verification routine comprises executable instructions for verifying the integrity of the executable program, as described in more detail below. The verification routine is inserted 6 into the executable program by appending the routine to the end of the executable program or by

embedding the routine within the executable program at one or more locations consisting of redundant code. Methods of identifying redundant code within the executable program are described below.

The locations of direct call instructions (as opposed to indirect call instructions) within the selected functional blocks are then identified 7 by reviewing the contents of the debug database, which stores the locations of direct call instructions and the addresses to which the call instructions point.

For each identified call instruction, respective intermediary code is inserted 8 into the executable program. As with the verification routine, the intermediary code may be appended to the end of the executable program or embedded at locations consisting of redundant code.

Each segment of intermediary code comprises two call instructions. The first call instruction points to the start address of the verification routine. The second call instruction points to the same address as that of the call instruction to which the intermediary code corresponds.

After the intermediary code of a respective call instruction is inserted 8, the call instruction of the functional block is modified 9 such that the call instruction points to the start address of the intermediary code.

Figure 3(a) illustrates an unprotected executable program 10 and Figure 3(b) illustrates a protected executable program 11 following insertion of protection code 12. The executable program 10 comprises four functional blocks 13. The second functional block 13b comprises a call instruction that points to the fourth functional block 13d. After insertion of the protection code 12, the call instruction of the second function block 13b points to respective intermediary code 14. The intermediary code 14 comprises two call instructions 15,16, the first call

instruction 15 points to a verification routine 17 and the second call instruction 16 points to the fourth functional block 13d.

When the protected executable program 11 is executed, and a call is subsequently made to the second functional block 13b, the execution thread calls the intermediary code 14. The execution thread then executes the first instruction 15 of the intermediary code 14 and calls the verification routine 17, which verifies the integrity of the protected executable program 11. After execution of the verification routine 17, the execution thread returns to the intermediary code 14, executes the second call instruction 16 and calls the fourth functional block 13d. After execution of the fourth functional block 13d, the execution thread returns to the address immediately after the call instruction of the second functional block 13b, whereupon the remaining instructions of the second functional block 13b are executed. Finally, the return instruction of the second functional block 13b is executed, causing the execution thread to return to the originating call instruction.

The verification routine 17 verifies the integrity of the protected executable program 11 by performing one or more redundancy checks. Two different types of redundancy check may be performed by the verification routine 17. In the first, a static redundancy check is performed on the protected executable program 11 stored on disk. In the second, a dynamic redundancy check is performed on the protected executable program held in memory during execution.

During the step of inserting 4 protection code 12 into the executable program 10, one or more CRC values, or other redundancy check data, are derived from the executable program. For a static redundancy check, a single CRC value is derived from the full code of the protected executable program. For a dynamic redundancy check, each CRC value is derived from a fragment of code of the executable program. The CRC value or values are then stored within the

protected executable program 11 (e.g. at locations within the verification routine 17) for subsequent use by the verification routine 17.

During subsequent execution of the protected executable program 11, the verification routine 17 recalculates the CRC value or values of the executable program 11 and compares these against the previously stored values in order to verify whether corruption of the executable program 11 has occurred. Any malicious code (e.g. debug code, virus code and phishing code) inserted into the executable program 11 will therefore be immediately identified.

If the verification routine 17 detects corruption of the protected executable program 11, the verification routine 17 terminates execution of the executable program 11. Optionally, the verification routine 17 corrupts or deletes any code of the executable program 11 stored in memory so as to prevent instructions being dumped from memory to disk.

Advantageously, the verification routine 17 comprises instructions for performing both a static redundancy check and a dynamic redundancy check. Alternatively, separate verification routines for performing a static redundancy check and a dynamic redundancy check may be inserted into the executable program. A call instruction of a functional block at or near the beginning of the executable program is then preferably modified such that a static redundancy check is performed early in the execution of the executable program. Consequently, any malicious code resident in the protected executable program may be identified before it is executed.

In the embodiment described above, intermediary code is inserted for each call instruction of the selected functional blocks, and each call instruction is modified to point to respective intermediary code. However, it is not essential that each and every call instruction of the functional blocks be protected in this way.

Instead, one or more call instructions of the functional blocks may be left unchanged by the protection process.

Each segment of intermediary code comprises two call instructions. The location of two neighbouring call instructions may provide a target for hackers. Accordingly, the intermediary code may include additional defunct code between the two call instructions. For example, the intermediary code may include a jump instruction and junk code sandwiched between the two call instructions, the jump instruction pointing to the second call instruction.

In the embodiment described above, each segment of intermediary code 14 comprises a call instruction 15 to the verification routine 17. Alternatively, however, the call instruction 15 of one or more segments of intermediary code may be replaced a respective verification routine. The instructions of the verification routine may be different for each segment of intermediary code. Consequently, should a hacker manage to identify the location of one verification routine, the hacker would not necessary be capable of identifying other verification routines.

In the embodiment described above, the verification routine 17 performs CRC checks to detect possible corruption of the protected executable program. Additionally or alternatively, the verification routine 17 may comprise instructions for determining whether debug processes are executing and/or whether debug or reverse engineering tools are installed.

It will be understood that, in this embodiment, the protection code 12 inserted into the executable program comprises at least one verification routine 15 and two call instructions. The first call instruction points to the verification routine (either directly or indirectly by means of an intermediary call instruction) and is inserted into the executable program so as to overwrite an existing call instruction. The second call instruction corresponds to the overwritten call

instruction and is inserted into the executable program such that it is executed after the first call instruction.

In a second embodiment, as illustrated in Figures 4, the step of inserting 4 protection code into the executable program comprises encoding 20 the instructions of a selected functional block with an encoding algorithm to create encoded instructions, inserting 21 a decoding routine into the executable program, and replacing 22 the functional block with a protected functional block, the protected functional block comprising a call instruction to the decoding routine and the encoded instructions. Each of the selected functional blocks is encoded in a similar manner and replaced by a respective protected functional block.

The 'encoding algorithm' comprises any algorithm capable of reversibly transforming the instructions of the functional block into a non-executable form. In particular, the terms 'encoding' and 'decoding' are intended to include, among other things, encryption and decryption as well as compression and decompression.

The decoding routine includes instructions for decoding the encoded instructions and executing the decoded instructions. The decoding routine uses the return address of the call instruction to identify the location of the encoded instructions within the protected block.

The decoding routine may additionally include a verification routine for verifying the integrity of the executable program, as described above. The verification routine optionally precedes any instructions for decoding such that decoding occurs only in the event that no corruption of the executable program is detected.

A functional block of an executable program may include a jump or call instruction to an address within a further functional block. If this further functional block comprises encoded instructions, a fatal error will arise. Consequently, not all functional blocks of the executable program are necessarily suitable for protection by encoding. Accordingly, the method additionally comprises reviewing the contents of the debug database to determine whether any of the direct call and jump instructions listed therein point to a location within a selected functional block. If a call or jump instruction points to a location within a selected functional block, protection of the functional block by encoding is prevented. If, however, a call or jump instruction points to the start address of the functional block, protection of the functional block by encoding is permitted.

The size of a protected functional block (i.e. the call instruction and encoded instructions) may be greater than the size of the original, unprotected functional block. Consequently, if the original functional block were replaced with the protected block, the encoded instructions would overwrite instructions of the subsequent functional block. Several methods may be employed to avoid overwriting a subsequent functional block.

In a first method, the selected functional block is examined for redundant code, using methods described below. Any redundant code is then omitted from the encoding process such that only useful instructions are encoded. By omitting redundant code from the encoding process, the size of the protected functional block is reduced.

In a second method, the protected functional block is divided into two or more fragments. The original functional block is overwritten by a first fragment of the protected functional block, which includes at least the call instruction. The remaining fragments of the protected functional block are then inserted (i.e. appended or embedded) into the executable program at other locations. Each fragment of the protected functional block may terminate with a pointer to the

address of the next fragment such that the decoding routine is able to retrieve all fragments of the protected functional block. Alternatively, the first fragment of the protected functional block may include a list of the addresses of all remaining fragments of the protected functional block.

Third, the protection of a functional block may be prohibited in the event that the corresponding protected functional block is of a greater size.

Figure 5(a) illustrates an unprotected executable program 10 and Figure 5(b) illustrates a protected executable program 11 following insertion of protection code 12. The executable program 10 comprises four functional blocks 13, the second 13b of which is selected for protection. The second functional block 13b of the protected executable program 11 has been replaced with a protected functional block 23, and a decoding routine 24 has been inserted into the executable program 11. The protected functional block 23 comprises a call instruction 25 to the decoding routine 24, and encoded instructions 26.

When the protected executable program 11 is executed, and a call is subsequently made to the protected functional block 23, the execution thread calls the decoding routine 24, which decodes the encoded instructions 26 of the protected functional block 23 and executes the decoded instructions. The final instruction of the decoded instructions comprises a return instruction, causing the execution thread to return to the originating call instruction.

In the embodiment described above, each of the selected functional blocks is encoded using the same encoding algorithm. Alternatively, different encoding algorithms may be used to encode different functional blocks, with a corresponding decoding routine for each encoding algorithm being inserted into the executable program. Different strengths of encoding can then be used according to the importance of the functional block being encoded. For example, a simple XOR function may be used for functional blocks of little importance,

whilst a strong encryption algorithm (e.g. DES, AES) may be used to encode more important functional blocks.

Encoding 20 the instructions of a functional block may comprise encrypting the instructions with an encryption algorithm that employs an encryption key. The encryption key may be derived from a fragment of code of the executable program. For example, a CRC value may be calculated for a fragment of the executable program and used to form an encryption key. The decoding routine then comprises instructions for calculating a CRC value for the same fragment of the executable program to form the decryption key. Consequently, the encoded instructions of the protected functional block are successfully decrypted only in the event that the fragment of the executable program from which the decryption key has been derived has not been corrupted.

In a further alternative, the encryption key used to encrypt a particular functional block may be derived from the instructions of a further (e.g. the immediately preceding) functional block, which is then itself encrypted. Consequently, a protected functional block is successfully decrypted only in the event that the further functional block is also successfully decrypted.

It will be understood that, in this embodiment, the protection code 12 inserted into the executable program 10 comprises at least one decoding routine 24 and at least one protected functional block 23, the protected block 23 being inserted into the executable program 10 so as to overwrite an existing functional block 13b. The protected functional block 23 comprises a call instruction 25 to the decoding routine 24 and encoded instructions 26, the encoded instructions 26 corresponding to those instructions of the functional block 13b that are overwritten.

Reference has thus far been made to encoding all instructions of a selected functional block. However, it may be preferable to encode only a selection of

instructions within a functional block. In particular, several small segments of encoded instructions interspersed with unencoded instructions may provide a more robust form of protection than encrypting all instructions of the functional block. Additionally, it may not be suitable or practical to encode particular instructions of a functional block. Accordingly, in an alternative embodiment, only selected instructions of the functional block are encoded. This then provides coarse (encoding all instructions) and fine (encoding only selected instructions) control of encoding.

In order to select instructions for encoding, the functional block is first disassembled. Since the start address of the functional block is known from the debug database, disassembly of the functional block is much more reliable than it would otherwise have been had the start address not been known. From the resulting assembly code, instructions of the functional block are selected for encoding. The code of the executable program corresponding to the selected instructions is then encoded in the manner described above.

The two embodiments described above for inserting protection code need not be used exclusively but may be used in combination. For example, the same functional block may be subject to both of the described embodiments.

Reference has been made above to the insertion of code into the executable program at locations occupied by redundant code. There are at least two methods in which redundant code may be identified within the executable program.

In the first method, use is made of the debug database associated with the executable program. In addition to storing the start address of each functional block of the executable program, the debug database additionally stores the end address of each functional block and/or the size of each functional block. Redundant code between functional blocks may therefore be identified by

comparing the end address of one functional block and the start address of the adjacent functional block.

In the second method, each functional block is disassembled. Redundant code is then identified from the assembly code as a repetitive series of instructions, e.g. no-op instructions, or patterns of known compiler / linker padding instructions.

The method embodying the present invention may be implemented as a suite of one or more computer programs. The program suite reads the contents of the debug database associated with the executable program to be protected, and displays (or otherwise outputs) a hierarchical listing of the functional blocks of the executable program. From this listing, the user is able to browse and select one or more functional blocks to be protected. An option is preferably provided to enable the user to select a functional block for disassembly. If a functional block is selected for disassembly, the program suite disassembles the selected functional block and displays (or otherwise outputs) a listing or representation of the assembly code. From the assembly code listing or representation, the user selects instructions of the functional block that require protection. Additionally, the program suite optionally enables a user to select the method or methods of protection for a selected functional block (or portion of a functional block) and, if appropriate, the type and strength of encoding. Once the user has selected the functional blocks (or portions of functional blocks) to be protected, the program suite inserts the protection code into the relevant locations of the executable program to create a protected executable program.

With the method embodying the present invention, an executable program may be protected through the insertion of protection code into the operational workings of the program. This then inhibits reverse engineering and/or tampering of the program. By using a debug database to identify the locations of functional blocks as well as the addresses of call and jump instructions, executable

programs for which the source code is unknown may be protected. Moreover, specific areas of the executable program may be targeted for protection. In particular, sections of code of the executable program may be reliably disassembled and specific instructions identified and protected.

When used in this specification and claims, the terms "comprises" and "comprising" and variations thereof mean that the specified features, steps or integers are included. The terms are not to be interpreted to exclude the presence of other features, steps or components.

The features disclosed in the foregoing description, or the following claims, or the accompanying drawings, expressed in their specific forms or in terms of a means for performing the disclosed function, or a method or process for attaining the disclosed result, as appropriate, may, separately, or in any combination of such features, be utilised for realising the invention in diverse forms thereof.

CLAIMS

1. A method of protecting an executable program, the method comprising:

receiving a copy of the executable program, the executable program comprising executable instructions arranged as a plurality of functional blocks;

receiving a debug database associated with the executable program, the debug database storing the locations of the functional blocks within the executable program;

identifying from the debug database the location of a functional block within the executable program; and

inserting protection code into the executable program, the protection code overwriting at least part of the functional block,

wherein execution of the functional block causes the protection code to be executed, and execution of the protection code performs an operation and executes a copy of the at least part of the functional block overwritten by the protection code.

2. A method according to claim 1, wherein inserting protection code comprises:

encoding at least part of the functional block to create encoded instructions;

inserting a decoding routine into the executable program; and

replacing the at least part of the functional block of the executable program with a call instruction to the decoding routine and with the encoded instructions,

wherein the decoding routine comprises executable instructions for decoding the encoded instructions and for executing the decoded instructions.

3. A method according to claim 2, wherein encoding comprises encrypting the at least part of the functional block using an encryption key, and decoding comprises decrypting the encoded instructions using a decryption key.

4. A method according to claim 3, wherein the encryption key is derived from a fragment of code of the executable program, the decoding routine derives the decryption key from the fragment of code, and the encoded instructions are successfully decrypted in the event that no corruption of the fragment of code has occurred.

5. A method according to claim 4, wherein the encryption key is derived from the instructions of a further functional block, the further functional block being encoded, and the decoding routine decodes the further functional block and derives the decryption key from the instructions of the further functional block, and the encoded instructions are successfully decrypted in the event that the further functional block is successfully decrypted.

6. A method according to any one of claims 2 to 5, wherein at least one functional block includes a call or jump instruction to a location within a further function block, and the debug database stores the location to which each call or jump instruction of the executable program points, and the step of encoding a functional block comprises:

selecting at least part of a functional block for encoding;

identifying from the contents of the debug database jump and call instructions that point to a location within the selected at least part of the functional block; and

encoding the selected at least part of the functional block in the event that no jump and calls instructions are identified.

7. A method according to any one of the preceding claims, wherein inserting protection code comprises:

encoding at least part of each of a plurality of functional blocks to create a plurality of segments of encoded instructions, each segment corresponding to a respective functional block;

inserting a decoding routine into the executable program; and

replacing the at least part of each of the plurality of functional blocks with a call instruction to the decoding routine and with a respective segment of encoded instructions,

wherein the decoding routine comprises executable instructions for decoding and executing encoded instructions.

8. A method according to any one of the preceding claims, wherein inserting protection code comprises:

encoding at least part of a first functional block with a first encoding algorithm to create first encoded instructions;

encoding at least part of a second functional block with a second encoding algorithm to create second encoded instructions;

inserting a first decoding routine into the executable program;

inserting a second decoding routine into the executable program;

replacing the at least part of the first functional block with a call instruction to the first decoding routine and with the first encoded instructions; and

replacing the at least part of the second functional block of the executable program with a call instruction to the second decoding routine and with the second encoded instructions,

wherein the first decoding routine comprises executable instructions for decoding the first encoded instructions and for executing the first decoded instructions, and the second decoding routine comprises executable instructions for decoding the second encoded instructions and for executing the second decoded instructions.

9. A method according to claim 8, wherein the first encoding routine and second encoding routine are different.

10. A method according to any one of claims 2 to 9, wherein the method further comprises disassembling the functional block and selecting one or more instructions from the disassembled code, and wherein encoding at least part of the functional block comprises encoding one or more fragments of the functional block corresponding to the selected instructions.

11. A method according to any one of claims 2 to 10, wherein the decoding routine is inserted into the executable program at locations occupied by redundant instructions.

12. A method according to claim 11, wherein the debug database stores the start address of each functional block and information regarding the end address of each functional block, and the method comprises comparing the end address of a functional block with the start address of an adjacent functional block to identify redundant instructions between the two functional blocks.

13. A method according to claim 11, wherein the method comprises disassembling a functional block, and analysing the disassembled code to identify redundant instructions within the functional block.

14. A method according to any preceding claim, wherein the functional block includes at least one call instruction having a call address, and the method further comprises:

inserting a verification routine into the executable program; and

modifying the call address of the call instruction such that the verification routine is executed prior to calling the call address.

15. A method according to claim 1, wherein the debug database stores the locations of call instructions within the executable program and inserting protection code comprises:

inserting a verification routine into the executable program;

identifying from the debug database the location of a call instruction within the executable program, the call instruction having a call address; and

modifying the call address of the call instruction such that the verification routine is executed prior to calling the call address.

16. A method according to claim 14 or 15, wherein the verification routine comprises instructions for detecting corruption of the executable program.

17. A method according to claim 16, wherein the verification routine comprises instructions for calling the call address in the event that no corruption is detected.

18. A method according to any one of claims 14 to 17, wherein at least a fragment of the executable program is stored in a memory device during execution and the verification routine detects corruption of the fragment stored in memory.

19. A method according to claim 18, wherein the verification routine comprises instructions for corrupting or deleting the fragment of the executable program stored in the memory device in the event that corruption is detected.

20. A method according to any one of claims 14 to 19, wherein the verification routine comprises instructions for terminating execution of the executable program in the event that the corruption is detected.

21. A method according to any one of claims 14 to 20, wherein the verification routine employs a CRC algorithm.

22. A method according to any one of claims 14 to 21, wherein the verification routine comprises instructions for detecting whether a debug process is executing and for terminating execution of the executable program in the event that a debug process is detected.

23. A method according to any one of claims 14 to 22, wherein the method further comprises inserting intermediary code into the executable program, the intermediary code comprising a first call instruction to the verification routine and a second call instruction to the call address, and the call address is modified to call the intermediary code.

24. A method according to any one of claims 14 to 23, wherein the method comprises identifying from the debug database the location of a plurality of call instructions within the executable program, each of the plurality of call instructions having a call address;

inserting a respective segment of intermediary code for each of the plurality of call instructions, each segment of intermediary code comprising a first call instruction to the verification routine and a second call instruction to the call address of a respective call instruction of the executable program; and

modifying the call address of each of the plurality of call instructions of the executable program to call a respective segment of intermediary code.

25. A method of protecting an executable program, the method comprising:

receiving a copy of the executable program, the executable program comprising executable instructions arranged as a plurality of functional blocks;

receiving a debug database associated with the executable program, the debug database storing the locations of the functional blocks within the executable program;

identifying from the debug database the location of a functional block within the executable program;

encoding at least part of the functional block to create encoded instructions;

inserting a decoding routine into the executable program; and

replacing the at least part of the functional block of the executable program with a call instruction to the decoding routine and with the encoded instructions,

wherein the decoding routine comprises executable instructions for decoding the encoded instructions and for executing the decoded instructions.

26. A method of protecting an executable program, the method comprising:

receiving a copy of the executable program;

receiving a debug database associated with the executable program, the debug database storing the locations of call instructions within the executable program;

inserting a verification routine into the executable program;

identifying from the debug database the location of a call instruction within the executable program, the call instruction having a call address; and

modifying the call address of the call instruction such that the verification routine is executed prior to calling the call address.

27. A device for protecting an executable program, the device comprising:

means for receiving a copy of the executable program, the executable program comprising executable instructions arranged as a plurality of functional blocks;

means for receiving a debug database associated with the executable program, the debug database storing the locations of the functional blocks within the executable program;

means for identifying from the debug database the location of a functional block within the executable program; and

means for inserting protection code into the executable program, the protection code overwriting at least part of the functional block,

wherein execution of the functional block causes the protection code to be executed, and execution of the protection code performs an operation and executes a copy of the at least part of the functional block overwritten by the protection code.

28. A device for protecting an executable program, the device comprising:

means for receiving a copy of the executable program, the executable program comprising executable instructions arranged as a plurality of functional blocks;

means for receiving a debug database associated with the executable program, the debug database storing the locations of the functional blocks within the executable program;

means for identifying from the debug database the location of a functional block within the executable program;

means for encoding at least part of the functional block to create encoded instructions;

means for inserting a decoding routine into the executable program; and

means for replacing the at least part of the functional block of the executable program with a call instruction to the decoding routine and with the encoded instructions,

wherein the decoding routine comprises executable instructions for decoding the encoded instructions and for executing the decoded instructions.

29. A device for protecting an executable program, the device comprising:

means for receiving a copy of the executable program;

means for receiving a debug database associated with the executable program, the debug database storing the locations of call instructions within the executable program;

means for inserting a verification routine into the executable program;

means for identifying from the debug database the location of a call instruction within the executable program, the call instruction having a call address; and

means for modifying the call address of the call instruction such that the verification routine is executed prior to calling the call address.

30. A computer comprising instructions for performing a method according to any one claims 1 to 26.

31. A computer program or suite of computer programs executable by a computer to perform a method according to any one of claims 1 to 26.

32. A computer program product storing computer program code executable by a computer to perform a method according to any one of claims 1 to 26.

33. A method substantially as hereinbefore described with reference to and as shown in the accompanying figures.

34. A device operable to perform a method substantially as hereinbefore described with reference to and as shown in the accompanying figures.

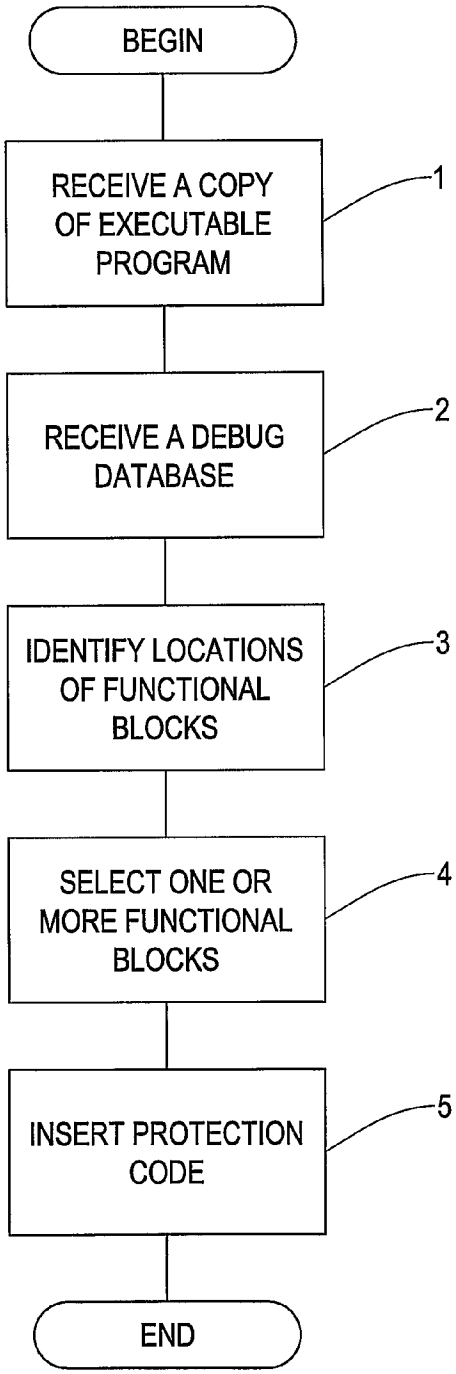

BEGIN
RECEIVE A COPY OF EXECUTABLE PROGRAM
1
RECEIVE A DEBUG DATABASE
2
IDENTIFY LOCATIONS OF FUNCTIONAL BLOCKS
3
SELECT ONE OR MORE FUNCTIONAL BLOCKS
4
INSERT PROTECTION CODE
5
END

FIG. 1

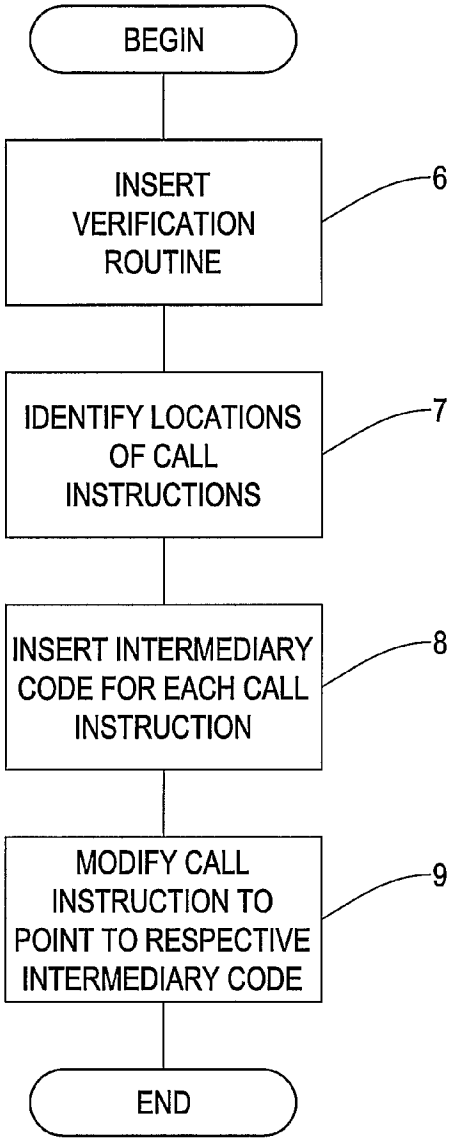

BEGIN
INSERT VERIFICATION ROUTINE
6
IDENTIFY LOCATIONS OF CALL INSTRUCTIONS
7
INSERT INTERMEDIARY CODE FOR EACH CALL INSTRUCTION
8
MODIFY CALL INSTRUCTION TO POINT TO RESPECTIVE INTERMEDIARY CODE
9
END

FIG. 2

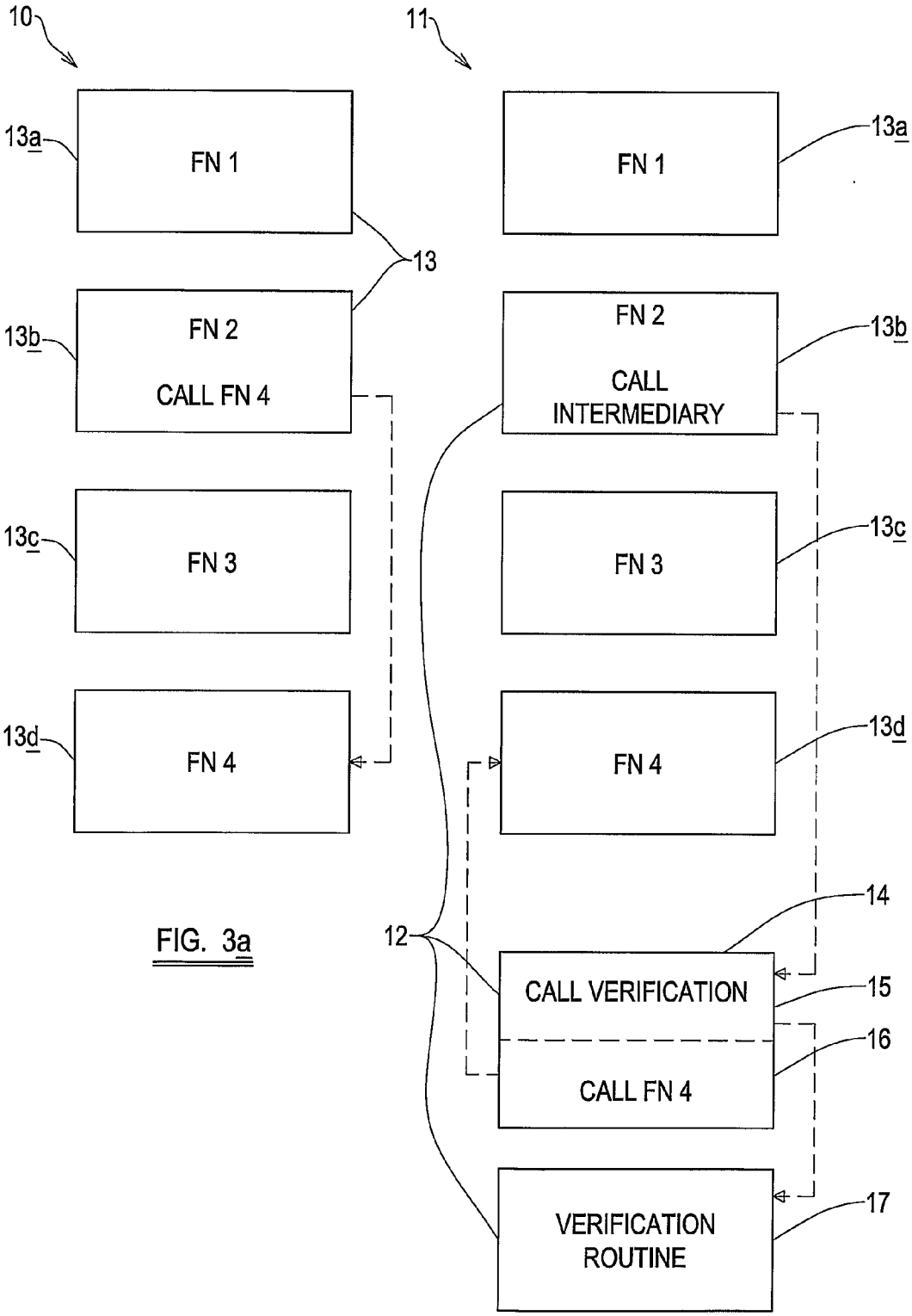

10
13a
FN 1
13
13b
FN 2
CALL FN 4
13c
FN 3
13d
FN 4

FIG. 3a

11
FN 1
13a
FN 2
CALL
INTERMEDIARY
13b
FN 3
13c
FN 4
13d
12
14
CALL VERIFICATION
15
16
CALL FN 4
VERIFICATION
ROUTINE
17

FIG. 3b

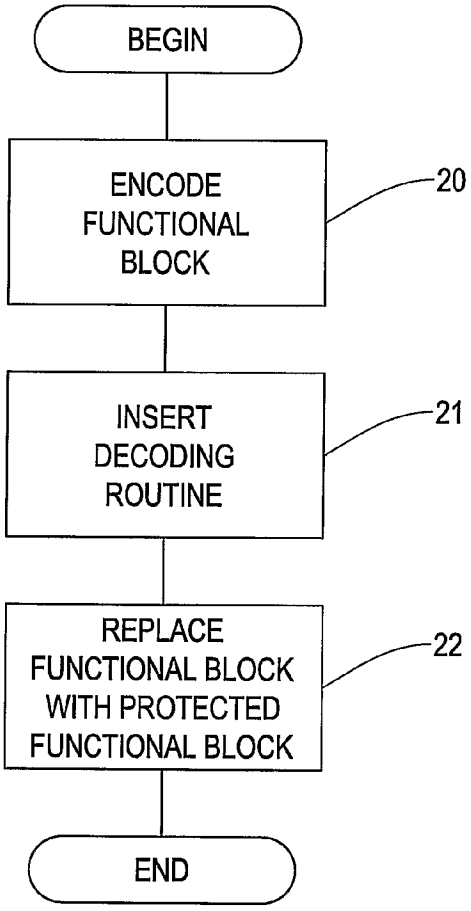

BEGIN
ENCODE FUNCTIONAL BLOCK
20
INSERT DECODING ROUTINE
21
REPLACE FUNCTIONAL BLOCK WITH PROTECTED FUNCTIONAL BLOCK
22
END

FIG. 4

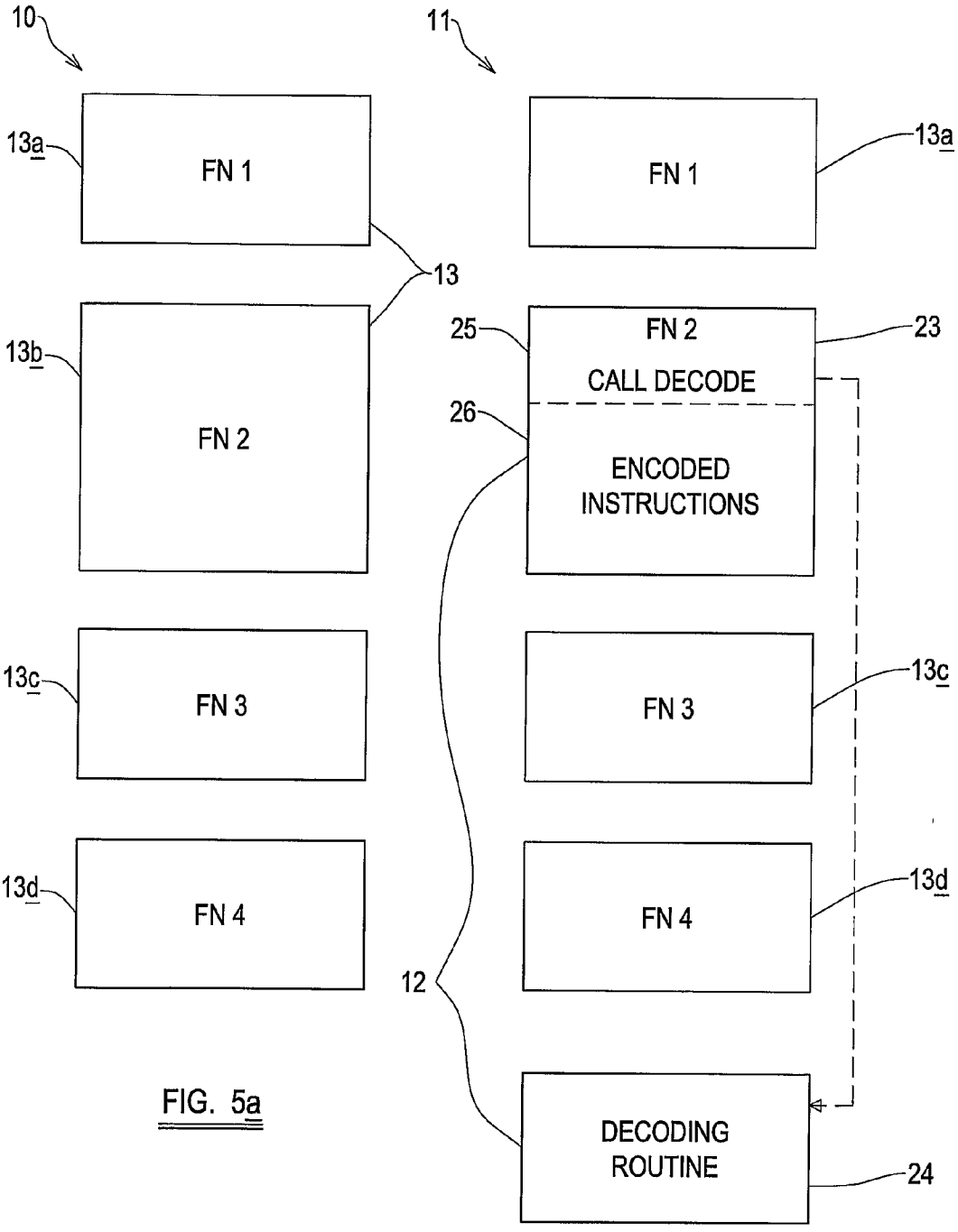

10
13a
FN 1
13b
FN 2
13c
FN 3
13d
FN 4
13
11
FN 1
13a
25
FN 2
CALL DECODE
23
26
ENCODED
INSTRUCTIONS
FN 3
13c
FN 4
13d
12
DECODING
ROUTINE
24

FIG. 5a

FIG. 5b

INTERNATIONAL SEARCH REPORT

International application No
PCT/GB2006/000483

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06F21/22

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)
EPO-Internal

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X Y	US 2004/003264 A1 (ZEMAN PAVEL ET AL) 1 January 2004 (2004-01-01) abstract figures 1,3,7-10 paragraph [0001] - paragraph [0008] paragraph [0025] - paragraph [0033] paragraph [0050] - paragraph [0089] ---	1,2,7, 10,25-32 3-6,8,9, 11-24
Y A	US 6 405 316 B1 (KRISHNAN GANAPATHY ET AL) 11 June 2002 (2002-06-11) abstract; figures 1-14 column 1, line 13 - column 17, line 14 claims 1-7 ---	3-6,8,9, 11-24
	-/--	

[X] Further documents are listed in the continuation of Box C.

[X] See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
18 May 2006	31/05/2006

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31–70) 340–2040, Tx. 31 651 epo nl, Fax: (+31–70) 340–3016	Kleiber, M

INTERNATIONAL SEARCH REPORT

International application No
PCT/GB2006/000483

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	WO 03/009114 A (LIQUID MACHINES, INC) 30 January 2003 (2003-01-30) page 3, line 2 - page 7, line 25 page 13, line 3 - page 36, line 19	1-32
A	US 2003/093685 A1 (TOBIN JOHN P.E) 15 May 2003 (2003-05-15) abstract figures 3-11 paragraph [0009] - paragraph [0010] paragraph [0024] - paragraph [0065]	1-32

FURTHER INFORMATION CONTINUED FROM PCT/ISA/ 210

The applicant's attention is drawn to the fact that claims relating to inventions in respect of which no international search report has been established need not be the subject of an international preliminary examination (Rule 66.1(e) PCT). The applicant is advised that the EPO policy when acting as an International Preliminary Examining Authority is normally not to carry out a preliminary examination on matter which has not been searched. This is the case irrespective of whether or not the claims are amended following receipt of the search report or during any Chapter II procedure. If the application proceeds into the regional phase before the EPO, the applicant is reminded that a search may be carried out during examination before the EPO (see EPO Guideline C-VI, 8.5), should the problems which led to the Article 17(2) declaration be overcome.

INTERNATIONAL SEARCH REPORT

International application No.
PCT/GB2006/000483

Box II Observations where certain claims were found unsearchable (Continuation of item 2 of first sheet)

This International Search Report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. ☐ Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely:

2. ☐ Claims Nos.:
because they relate to parts of the International Application that do not comply with the prescribed requirements to such an extent that no meaningful International Search can be carried out, specifically:

see FURTHER INFORMATION sheet PCT/ISA/210

3. ☐ Claims Nos.:
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

Box III Observations where unity of invention is lacking (Continuation of item 3 of first sheet)

This International Searching Authority found multiple inventions in this international application, as follows:

1. ☐ As all required additional search fees were timely paid by the applicant, this International Search Report covers all searchable claims.

2. ☐ As all searchable claims could be searched without effort justifying an additional fee, this Authority did not invite payment of any additional fee.

3. ☐ As only some of the required additional search fees were timely paid by the applicant, this International Search Report covers only those claims for which fees were paid, specifically claims Nos.:

4. ☐ No required additional search fees were timely paid by the applicant. Consequently, this International Search Report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:

Remark on Protest

☐ The additional search fees were accompanied by the applicant's protest.

☐ No protest accompanied the payment of additional search fees.

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No
PCT/GB2006/000483

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2004003264 A1	01-01-2004	NONE	
US 6405316 B1	11-06-2002	AU 6051798 A	18-08-1998
		WO 9833106 A1	30-07-1998
		US 6141698 A	31-10-2000
WO 03009114 A	30-01-2003	EP 1410150 A2	21-04-2004
US 2003093685 A1	15-05-2003	NONE	